EXTERNAL SEMANTIC QUERY SIMPLIFICATION: A GRAPH-THEORETIC APPROACH AND ITS IMPLEMENTATION IN PROLOG

Matthias Jarke

June 1984

Center for Research on Information Systems
Computer Applications and Information Systems Area
Graduate School of Business Administration
New York University

Working Paper Series

CRIS #75

GBA #84-51(CR)

Published in Proceedings of First International Workshop on Expert Database Systems, Kiawah Island, South Carolina, October 1984, pp.467-482.

EXTERNAL SEMANTIC QUERY SIMPLIFICATION: A GRAPH-THEORETIC APPROACH AND ITS IMPLEMENTATION IN PROLOG

Abstract

Semantic query simplification utilizes integrity constraints enforced in a database system for reducing the number of tuple variables and terms in a relational calculus query. To a large degree, this can be done by a system that is external to the DBMS. The paper advocates the application of database theory in such a system and describes a working prototype of an external semantic query simplifier implemented in Prolog. The system employs a graph-theoretic approach to integrate tableau techniques and algorithms for the syntactic simplification of queries containing inequality conditions. The use of integrity constraints is shown not only to improve efficiency but also to permit more meaningful error messages to be generated, particularly in the case of an empty query result. The paper concludes with outlining an extension to the multi-user case.

1.0 INTRODUCTION

A research project at New York University [Jarke and Vassiliou 1984; Vassiliou et al. 1983] investigates the integration of logic-based expert systems into existing management information systems. Several prototype expert systems in life insurance [Jarke and Sivasankaran 1984] and management science are being built which rely heavily on access to large databases containing, e.g., model input, actuarial data, customer data, or health scoring information.

The interaction between expert systems and existing databases requires coupling two independent software systems: the expert system, e.g., written in Prolog, and a database system accessible through a relational query language, e.g., SQL. Rather than writing application-specific access routines as customary in the expert systems area, it was decided to build a generalized software tool that provides information to the expert system as and when required for the expert's deduction, much in the same way a human expert might consult a database for certain facts [Vassiliou et al. 1984; Jarke et al. 1984].

While the original motivation for building such a tool was its use as a data management backend to an expert system, it is not hard to see that the other direction of interaction is at least equally desirable. Very high-level user interfaces to databases make use of deductive components but often lack an efficient interface between these components and an existing database. So-called deductive database systems partially solve this problem but stress a very deep integration with the underlying database (BDGEN [Nicolas and Yazdanian

1983]) or attempt to build one integrated system (e.g., DADM [Kellogg 1982]). In contrast, our approach assumes independent existing systems and attaches the translation procedure to the expert systems language rather than to the DBMS (which may be used for many other purposes in addition to its use as an expert system backend).

A second aspect of enhancing DBMS with semantic knowledge has been worked upon to a lesser degree so far: the knowledge-based execution of conventional database operations [Hammer and Zdonik 1980; King 1981]. Current DBMS are typically good in evaluating alternative strategies for processing a query on the physical level. They are often less strong in transforming a query submitted by the user into a (possibly different) representation which lends itself to the creation of more efficient processing alternatives, in particular when queries to views are concerned [Ott and Horlaender 1982]. Moreover, processing a sequence or set of related queries is rarely supported.

A coupling mechanism allows the creation of an 'expert system' <u>external to</u> the DBMS that might employ syntactic and semantic knowledge about the database schema, as well as about strengths and weaknesses of the query optimizer of the underlying DBMS, to rephrase and organize a query or set of queries in the most efficient way. While in theory inferior to a fully integrated intelligent DBMS query optimizer (which would have full access to all internal data structures and full information about the database state at any given time), such an external 'database programming expert' may well benefit many existing databases in which the code of the DBMS is not accessible or should not be touched for reliability reasons.

The purpose of the present paper is twofold. Firstly, it tries to clarify the concept of semantic query simplification, as compared to other approaches to utilizing general laws (or AI rules) in DBMS. In particular, it is argued that results obtained by database theory research should be employed as a crucial part of the knowledge bases and inference mechanisms in knowledge-based query evaluation methods although there is additional knowledge that has to be captured and utilized in a less structured manner, both in the application domain and in the query optimization domain itself.

Secondly, the paper reports preliminary experience with a working prototype of a semantic query simplifier implemented in Prolog whose knowledge base may contain key dependencies, general functional dependencies, certain types of domain and inclusion dependencies, and some 'expert' rules added to the system to reduce optimization time (although these may in rare cases prevent optimality of the result). An overall algorithm has been described in [Jarke et al. 1984]. This paper presents a more efficient, integrated method that is based on a graph-theoretic representation of tableau techniques and handles arbitrary conjunctive queries with inequalities. The paper concludes with an outline of extensions currently under study, in particular with the concept of a multi-user querying front end.

2.0 SEMANTIC QUERY OPTIMIZATION, DATABASE THEORY, AND PROLOG

Rules in database systems. While the main purpose of most current DBMS is the management of large amounts of formatted specific facts, some DBMS support general rules that govern which data can be stored in the database (integrity constraints) or how to derive new facts from the stored ones; the latter are called deduction rules if applied at query time and generation rules if used to store derived facts explicitly. In other words, deduction and generation rules increase the number of facts retrievable from the database beyond the originally inserted facts, whereas integrity constraints reduce the number of facts that can be stored and retrieved.

Semantic query optimization employs integrity constraints for transforming queries, in the extreme to the degree that they can be answered without looking at the stored facts at all. The underlying principle of semantic query optimization is that one can add to each query predicate, P, an arbitrary number of integrity constraints, C1, ..., Cn, to form a new predicate:

P AND C1 AND ... AND Cn

without changing the result of the query (since all integrity constraints are always true by definition). The new predicate can then be converted -- by syntactic transformations (e.g., idempotency laws of the relational calculus [Jarke and Koch 1983]) -- into a form that lends itself to more efficient evaluation. We speak of semantic query simplification if the query resulting from this process never has more terms or tuple variables in its predicate than the original one. This will be the case if a subpredicate of P is implied by the added integrity constraints (i.e., the subpredicate is redundant and can be omitted) or contradicts them (i.e., the query result will be empty by definition). Interestingly, the basic ideas underlying this kind of optimization appeared almost simultaneously in a database theory [Aho et al. 1979] and in an AI context [Hammer and Zdonik 1980; King 1981]. However, it seems that the connection between the two approaches has not generally been recognized. Demonstrating the practicality of this relationship is one of the goals of the present paper.

Knowledge bases for semantic query optimization. A major issue in semantic query optimization has been the reduction of the search space for applicable integrity constraints and efficiency-enhancing query transformations [King 1981; Du 1983]. It is our perception that the type of integrity constraints existing in the system has a substantial influence on how this reduction can best be achieved.

In particular, there may be a discrepancy between the scope of typical integrity constraints in a relational database system and in AI-based knowledge representation (e.g., a semantic net or a set of Prolog view definitions). With few exceptions (e.g., [Klug 1980]), database theory has concentrated on those types of general laws that are applicable to all elements of one relation (e.g., domain or functional dependencies), or to a combination of relations (e.g., inclusion dependencies). It is therefore (relatively) easy to recognize the applicability of an integrity constraint to a particular query, and to develop powerful -- sometimes provably optimal -- inference mechanisms. The task may be further simplified by the fact that the same laws are also used in the database design process to structure the database.

In contrast, published work in semantic query optimization has focused on more specific constraints that capture chunks of knowledge about smaller sets of data; simple examples of such constraints include: "only tankers have more than 400,000 tdw" [King 1981] or "assistant professors do not have tenure". Here, it is not sufficient to look at a relation name in the constraint definition, since the applicability of a constraint to a certain tuple depends on membership in a subrelation. Moreover, the number of constraints is potentially very large and tends to be a function of the number of tuples (database size) rather than of the number of attributes (schema size). Finally, it is often not clear whether, how, and to what degree the addition of an integrity constraint will improve the efficiency of query evaluation: the resulting query may contain fewer or more terms than the old one. Artificial Intelligence-type heuristics and information about the database state at query execution time are frequently required for making these decisions.

Ultimately, the feasible extent of semantic query optimization depends on two factors: (a) what types of integrity constraints are enforced by the DBMS? and (b) what amount of search for optimization strategies is justified by the expected savings in query execution time? For an external semantic query optimizer, the heavy reliance on database theory and generally applicable laws has the advantage that the number of integrity constraint definitions (to be kept consistent between the optimizer and the DBMS) is relatively small and that little knowledge is required about the current database state; the latter type of knowledge is assumed to be handled by the DBMS query optimizer (this distinguishes this approach from Warren's [1981] who duplicates DBMS functions in his optimizer). Additionally, although there is a trend towards more sophisticated integrity assertions [Blaustein 1980], most current database systems do not go beyond relatively simple concepts, such as bounds for numerical attribute values, key or at most general functional dependencies, and certain types of inclusion dependencies, e.g., unary ones [Cosmadakis and Kanellakis 1984] or referential constraints [Jarke et al. 1984].

As demonstrated in the sequel, these constraints can be employed quite efficiently in integrated query simplification algorithms that rely heavily on partial results provided by database theory. Such an algorithm has been implemented in DEC20-Prolog. Runtimes for a set of 14 test queries with four to six tuple variables and 5 to 20 join and restrictive terms were in the range between .5 and 1.2 seconds, including the times for translating from the Prolog form to the internal representation used by the optimizer, and from the optimized internal form to the DBMS query language. The usage of additional 'expert rules', obtained by observing systems behavior and intended to cut off less promising searches, even at the expense of guaranteed optimality, further reduces these times and, in particular, their growth rate with respect to the size of queries and the number of integrity constraints.

3.0 STRUCTURE OF THE SEMANTIC QUERY SIMPLIFIER

The semantic query simplifier consists of two translation mechanisms, a knowledge base, and the simplifier inference engine working on it, using a 'blackboard' [Erman and Lesser 1975] for intermediate results accessed and altered by multiple, largely independent algorithms (Figure 1). Thus, multiple 'experts' can be created for different kinds of integrity constraints.

```
    input query language (e.g., Prolog)
                      |
                      | METAEVALUATE
                      ↓
            dbcall language
                      |
                      | GENERATE_TABLEAU
                      ↓
      internal graph representation <------------------> knowledge
           on 'blackboard'                SIMPLIFY          base
                      |
                      | SQLTRANSLATE
                      ↓
     DBMS query language (e.g., SQL)
```

Figure 1: Structure of the External Semantic Query Simplifier

The knowledge base is specific to a particular database; it contains a schema definition and predicates describing the integrity constraints. The current system will utilize [1] key dependencies (one per relation), general functional dependencies (standardized so that they have only one attribute on the right-hand side), value bounds for numerical attributes, and referential integrity constraints, i.e., inclusion dependencies, in which the superset side must be a key and in which each attribute appears in at most one referential constraint on the subset side [Jarke et al. 1984]. Referential constraints were selected since they are central to the relational data model, yet have easier inference algorithms than general inclusion dependencies. Key dependencies have been implemented separately from other functional dependencies for three

[1] Additional constraints can be specified but will be ignored by the simplifier, since -- in this respect -- Prolog is purely declarative.

reasons. First, many systems support keys but much less handle general functional dependencies. Thus, being able to state key dependencies directly may be convenient for a user. Second, in tableau optimization, equal keys mean that two complete rows become equal and one of them can be removed, leading to the removal of one join operation in query evaluation. Finally, the use of key dependencies speeds up the simplification algorithm in comparison to the usual representation in which a functional dependency would have to be defined for each non-key attribute.

In summary, the knowledge base would be roughly appropriate for a database in Fagin's [1981] domain/key normal form, except that we allow the use of general functional dependencies. Figure 2 contains the Prolog description of the knowledge base for a two-relation database describing employees and their departments. There is a value bound on the salary attribute of the employee relation; note that the bounds could be defined either by the domain type, or they could represent the actual maximum and minimum value for the current database state if those are maintained [Blaustein 1980]. The two referential integrity constraints say that employees work only in departments that exist, and that managers are employees.

```
schema(employee,[eno, ename, salary, dno]).
keydep(employee, [eno]).
funcdep(employee, [ename], [eno]).
valuebound(employee, salary, 1000, 9000).

schema(department,[dno, dname, mgr]).
keydep(department, [dno]).
funcdep(department, [mgr], [dno]).

refint(employee, [dno], department, [dno]).
refint(department, [mgr], employee, [eno]).
```

<u>Figure 2</u>: Example of a knowledge base for the simplifier

The two translation mechanisms make the core simplifier more or less independent of its input (from the user) and output (to the DBMS) query languages. There are currently experimental interfaces for Prolog input [Vassiliou et al. 1984], and for relational algebra and SQL output. The simplifier itself expects its input in a tableau-like subset of Prolog [Jarke et al. 1984]. Essentially, each query is a list of "dbcall" predicates corresponding to the rows of a tableau:

```
dbcall(Relationname, List_of_tableau_entries)
```

or to the inequality comparisons:

```
dbcall(Operator, Left_operand, Right_operand)
```

where the operator may be one of: equal, notequal, lessequal, greaterequal, less, greater, and the operands are either domain variables appearing as tableau entries or constants. The simplifier does a limited amount of input checking by comparing the form of the input to the schema information, and constant values to the value bounds stored in the knowledge base. Domain variables are expected to be indicated syntactically by beginning with "t_" (for target variables) or with "v_" (for nondistinguished variables).

Figure 3 presents an example input query. If Prolog is used as the user query language, such queries are derived by processing deduction rules (view definitions) defined by Horn clauses [Vassiliou et al. 1984; Jarke et al. 1984]. For instance, the query in Figure 3 could have been derived from a view definition and Prolog query as given in Figure 4.

```
[dbcall(employee, [v_Eno1, t_X, v_Sal1, v_D]),
 dbcall(department, [v_D, v_Fct2, v_M]),
 dbcall(employee, [v_M, smiley, v_Sal3, v_Dno3]),
 dbcall(employee, [v_Eno, t_X, v_S, v_Dno]),
 dbcall(greaterequal, v_S, 4000),
 dbcall(lessequal, v_Sal1, v_Sal3),
 dbcall(lessequal, v_Sal3, 4000)]
```

Figure 3: An example dbcall query

```
/* example view definitions in Prolog: it is known that no manager
   makes more than 4000, but nobody makes more than his manager */

works_directly_for(X, Y) :-
      employee(Eno1, X, Sal1, D),
      department(D, Fct2, M),
      employee(M, Y, Sal3, Dno3),
      Sal1 =< Sal3,
      Sal3 =< 4000.

/* Prolog query: who works directly for smiley and makes at least 4000? */

:- works_directly_for(X, smiley), employee(Eno, X, S, Dno), S >= 4000.
```

Figure 4: Original Prolog query from which Figure 3 is generated by METAEVALUATE mechanism

The principle of the inference engine has been described in [Jarke et al. 1984]:

1. For each tableau variable that has a value bound constraint, add two inequalities to the query.

2. Set the Boolean variables REPEAT and FIRSTTIME to true.

3. Apply an inequality simplification algorithm; if a contradiction is detected, stop with an empty query result; if variables have to be renamed due to newly detected equality conditions or if FIRSTTIME, set REPEAT to true and FIRSTTIME to false, else set REPEAT to false.

4. If REPEAT then do the following: apply a functional dependency chase algorithm with deletion of duplicate rows; if a contradiction is detected, stop with an empty query result; if variables have been renamed return to 3.

5. Remove tableau rows that serve no other purpose than establishing the existence of certain tuples in a relation which can already be inferred from referential integrity constraints.

A shortcoming of this procedure is the complete separation of processing inequalities and functional dependencies which leads to substantial superfluous work. In the subsequent section, a new algorithm is described that integrates these two steps and results in less overall complexity (and real time savings, as shown by the comparison of the two implementations). In this method, a <u>blackboard</u> is used for managing predicates that are inserted for temporary, shared use by both subalgorithms and erased later. The overall algorithm always starts and ends with an 'clean' blackboard.

4.0 A GRAPH-BASED ALGORITHM AND ITS IMPLEMENTATION IN PROLOG

4.1 Two Graph Representations

The query simplifier uses two interacting graph representations: a query graph for representing a query containing inequalities, and an FD/KD graph for representing the application of functional and key dependencies. The former extends ideas by [Rosenkrantz and Hunt 1980] whereas the latter is based on concepts introduced in [Downey et al. 1980] who also proposed a fast congruence closure algorithm for determining the lossless join property of a tableau, a variation of which is used as part of the algorithm presented here. Both graphs share a common set of nodes but differ in edge semantics.

var-1 <= var-2	var-1 $\xrightarrow{0}$ var-2
var-1 < var-2	var-1 $\xrightarrow{-1}$ var-2
var-1 <= const	var-1 $\xrightarrow{\text{const}}$ 0(integer)
var-1 < const	var-1 $\xrightarrow{\text{const-1}}$ 0(integer)
var-1 >= var-2	var-1 $\xleftarrow{0}$ var-2
var-1 > var-2	var-1 $\xleftarrow{-1}$ var-2
var-1 >= const	var-1 $\xleftarrow{\text{-const}}$ 0(integer)
var-1 > const	var-1 $\xleftarrow{\text{-const-1}}$ 0(integer)

Figure 5: Construction of query graph from inequalities [Rosenkrantz and Hunt 1980]

The query graph is a labelled directed graph. The node set contains all entries appearing in the tableau (i.e., the dbcall predicates that reference relations), plus a node 0(d) for each ordered domain d [2]. Arcs represent inequality conditions. There are two types: those representing lessequal conditions, and those representing notequal conditions. Equality terms are handled by renaming; the remaining three operators are converted to lessequal arcs, as indicated in Figure 5.

[2] The current implementation allows only integer as this domain. Moreover, DEC20-Prolog allows only integers up to about +/- 131,000.

In Prolog, tableau element nodes are represented as 4-ary predicates asserted in the blackboard:

```
in_tableau(Tableau_entry, Level, Tableau_row_no, Attribute_name).
```

The latter two parameters characterize the position of a tableau entry in the input query. The Level parameter provides information when the entry was created with respect to the simplification process; it is necessary because the application of either of the two algorithms working on the blackboard can change tableau entries. Edges are represented by 5-ary predicates:

```
inequality(Identifier, Operator, Left_node, Right_node, Length).
```

where the Identifier is used for fast retrieval on the blackboard (e.g., for erasure or change of operand names) and Length is determined as indicated in Figure 5.

In the <u>FD/KD graph</u>, a bundle of directed edges connects each node whose attribute name appears on the right-hand side of a functional or key dependency in the knowledge base, to all the nodes corresponding to the left-hand side of that dependency. An example of a combined query and FD/KD graph is given in Figure 6 for the example in Figure 3. The FD/KD edges are not stored explicitly but derived when needed, utilizing Prolog's efficient pattern matching capabilities.

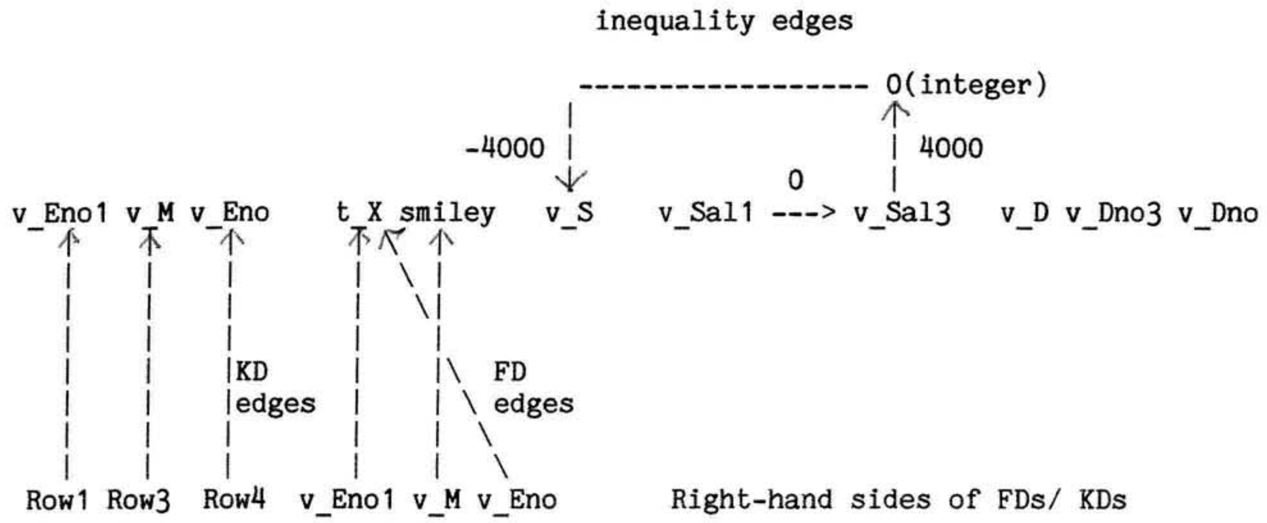
inequality edges
0(integer)
-4000
4000
0
v_Eno1 v_M v_Eno t_X smiley v_S v_Sal1 ---> v_Sal3 v_D v_Dno3 v_Dno
KD edges
FD edges
Row1 Row3 Row4 v_Eno1 v_M v_Eno
Right-hand sides of FDs/ KDs

Figure 6: A functional and key dependency graph overlayed with the inequality graph for the query example

4.2 Two Graph Algorithms And Their Integration

The two representations could now be used as in section 3 to simply implement a repeated execution of two separate algorithms until nothing changes any more. Instead, we shall first describe each of the algorithms and then present a better integration. The two algorithms below are extensions and adaptations of work by [Rosenkrantz and Hunt 1980] for the query graph, and by [Downey et al. 1980] for the FD/KD graph. They can be summarized as follows:

1. Inequality optimization: The algorithm can be summarized by the following Prolog rule:

```
process_inequalities :-
      remove_multiedges,
      compute_shortest_paths,
      postprocess_graph(0).
```

   Remove_multiedges succeeds after removing multiple redundant comparisons between any pair of nodes. A (deliberately extreme) example is given in Figure 7. Note, that the first inequality (greater, v_S, 200) is removed since it is implied by the valuebound on the salary attribute. (The output of the simplifier does not really have the same format as its input; the example has been translated back to the dbcall language for readability.)

```
/* a query with redundant inequality comparisons */
       [dbcall(employee, [v_Eno1, t_X, v_Sal1, v_D]),
        dbcall(department, [v_D, v_Fct2, v_M]),
        dbcall(employee, [v_M, v_Man, v_Sal3, v_Dno3]),
        dbcall(employee, [v_Eno, t_X, v_S, v_Dno]),
        dbcall(greater, v_S, 200),
        dbcall(equal, v_Man, smiley),
        dbcall(lessequal, v_S, 4000),
        dbcall(notequal, v_S, 6000),
        dbcall(notequal, v_S, 6000),
        dbcall(lessequal, v_S, 6000),
        dbcall(notequal, v_S, 4000)]

/* an equivalent query after removal of redundant inequalities */
       [dbcall(employee, [v_Eno1, t_X, v_Sal1, v_D]),
        dbcall(department, [v_D, v_Fct2, v_M]),
        dbcall(employee, [v_M, smiley, v_Sal3, v_Dno3]),
        dbcall(employee, [v_Eno, t_X, v_S, v_Dno]),
        dbcall(lessequal, v_S, 3999)]
```

Figure 7: Example for removal of multi-edges

Compute_shortest_paths creates, on the blackboard, a Prolog representation of the shortest paths between all pairs of nodes, using a simple algorithm of cubic (in the number of nodes) complexity, as described, e.g., in [Reingold et al. 1977]. The algorithm has been enhanced in the sense that it stops with an error message and an empty query result as soon as a negative length cycle (meaning 'A < A' for any node A on the cycle -- see the example in Figure 8) is detected, and that it considers only nodes that actually appear in inequalities.

```
| ?- query8(Q), generate_tableau(0,Q), process_inequalities.

warning: contradiction among inequalities

Q = [dbcall(employee,[v_Eno1,t_X,v_Sal1,v_D]),
     dbcall(department,[v_D,v_Fct2,v_M]),
     dbcall(employee,[v_M,v_Man,v_Sal3,v_Dno3]),
     dbcall(employee,[v_Eno,t_X,v_S,v_Dno]),
     dbcall(lessequal,v_S,4000),
     dbcall(greaterequal,v_Sal3,5000),
     dbcall(greater,v_S,v_Sal3)]

yes
```

$$\begin{tikzcd} v\_S \arrow[r, "4000"] & 0(integer) \arrow[r, "-5000"] & v\_Sal3 \arrow[ll, bend left, "-1"] \end{tikzcd}$$

Figure 8: Prolog log and query graph showing a contradiction between inequalities by a negative length cycle.

Postprocess_graph (the parameter corresponds to the previously mentioned Level parameter in the in_tableau predicates) follows the cycles with a total length of 0 and renames all variables appearing on such cycles, either to a single variable name or -- if any node 0(d) is on the cycle -- to a constant corresponding to the total length of the path from each node on the cycle to node 0(d). In the query graph, renaming leads to the removal of nodes and all their related arcs and shortest paths.

2. FD/KD optimization: A fast chase algorithm computes the congruence closure of the FD/KD graph in a breadth-first fashion, using the Level parameter to prescreen the tableau entries to which an FD or KD might be applicable at a given point in time. The algorithm terminates when, at a given level, there are no further in_tableau predicates with that level. In other words,

the algorithm tries first to apply all directly applicable FDs/KDs; afterwards, only such FDs/KDs can be applicable that have as their left-hand side tableau elements changed in the previous step. KDs are tried before FDs since their application leads to the deletion of a row and therefore renders the application of further FDs superfluous. As an example, consider the preprocessed query in Figure 7. At level 0, only one functional dependency is applicable, leading to the new query:

```
[dbcall(employee, [v_Eno1, t_X, v_Sal1, v_D]),
 dbcall(department, [v_D, v_Fct2, v_M]),
 dbcall(employee, [v_M, smiley, v_Sal3, v_Dno3]),
 dbcall(employee, [v_Eno1, t_X, v_S, v_Dno]),
 dbcall(lessequal, v_S, 3999)]
```

At level 1, the key dependency for the employee relation becomes applicable, leading to the deletion of the fourth row and to renaming of v_Sal1 to v_S in the first row. Another example is given in Figure 9; here, the notequal predicate prevents successful application of the key dependency and the query result will be empty.

```
| ?- query10(Q), generate_tableau(0,Q), simplify.

warning: contradiction by \= condition:
v_Dno cannot be equal to v_D
as required by a functional or key dependency

Q = [dbcall(employee,[v_Eno1,t_X,v_Sal1,v_D]),
     dbcall(department,[v_D,v_Fct2,v_M]),
     dbcall(employee,[v_M,smiley,v_Sal3,v_Dno3]),
     dbcall(employee,[v_Eno,t_X,4000,v_Dno]),
     dbcall(notequal,v_D,v_Dno)]

yes
```

Figure 9: Example of a contradiction detected by application of functional and key dependencies

A closer look at the interplay of these two algorithms shows that the results of each algorithm can be expressed in the notion of the other by integrating the two graph representations as shown in Figure 6; this in turn leads to a better integration that avoids full repetition of both algorithms at each stage of the algorithm given in section 3.

The most important observation concerns the application of a functional dependency by the second algorithm. Its result is that two tableau entries are made equal. If both entries, say X and Y, are variables, this corresponds to introducing zero-length edges from X to Y and from Y to X [3]. If previously there was a negative-length shortest path in either direction, this leads immediately to a negative length cycle and thus to a contradiction in the query. Otherwise, all of the shortest paths must be recomputed to look for new zero length cycles which could lead to variable renaming, using the postprocess_graph predicate at the current Level. However, the complexity of this recomputation is at most quadratic (rather than cubic as originally), since only each of the previous shortest paths has to be compared with a path through the new edges between X and Y.

For an example for the integrated procedure, consider again Figures 3 and 6. Adding zero length edges between v_S and v_Sal1 in Figure 6 through the application of a functional (level 0) and a key (level 1) dependency simplifies the query of Figure 3 to:

[3] If one entry (say Y) is a constant of domain d, the same procedure will follow but the edges to be added to the graph will be one from node X to node O(d) with length Y, and one from O(d) to X with length -Y. When X and Y are (different) constants, there is again a contradiction leading to a message and an empty query result.

```
     [dbcall(employee, [v_Eno1, t_X, 4000, v_D]),
      dbcall(department, [v_D, v_Fct2, v_M]),
      dbcall(employee, [v_M, smiley, 4000, v_Dno3])]).
```

Vice versa, changes in the tableau caused by the inequality algorithm will be indicated by the Level parameter of the in_tableau predicates on the blackboard, such that they can be exploited by the FD/KD algorithm in the same way as changes caused by previous FD/KD applications. The implementation of this interplay makes use of the recursion features of Prolog. A sketch of some of the high-level predicates follows (the system currently has about 200 clauses):

```
simplify :-
      process_inequalities,
      one_relation_simplify(0),
      remove_deletable_danglers.

one_relation_simplify(Level) :-
      rowrel(Row1, Rel), rowrel(Row2, Rel), Row2 > Row1,
      prescreen_and_simplify(Level, Row1, Row2, Rel),
      fail.
one_relation_simplify(Level) :-
      Level1 is Level + 1, in_tableau(_, Level1, _, _),
      !,
      one_relation_simplify(Level1).
one_relation_simplify(_).

prescreen_and_simplify(0, Row1, Row2, Rel) :-
      one_level_simplify(0, Row1, Row2, Rel), !.
prescreen_and_simplify(Level, Row1, Row2, Rel) :-
      (in_tableau(_, Level, Row1, _); in_tableau(_, Level, Row2, _)),
      !,
      one_level_simplify(Level, Row1, Row2, Rel).

one_level_simplify(Level, Row1, Row2, Rel) :-
      equal_key(Level, Row1, Row2, Rel), schema(Rel, Schema),
      !,
      coerce(Level, Schema, Row1, Row2),
      delete_row(Schema, Row2).
one_level_simplify(Level, Row1, Row2, Rel) :-
      equal_LHS(Level, Row1, Row2, Rel, RHS),
      coerce(Level, RHS, Row1, Row2),
      fail.
one_level_simplify(_, _, _, _).
```

The predicate, coerce, tries to make the values of the attributes in the list RHS equal between rows Row1 and Row2, gives appropriate error messages should this prove impossible due to contradictions, and indirectly activates the recomputation of shortest paths.

5.0 CONCLUSIONS AND EXTENSIONS

The practical relevance of tableau-oriented simplification techniques inspired by database theory has repeatedly been questioned by practioners, as evidenced by the fact that they are hardly implemented in any of the well-known relational systems. Our preliminary experience with an actual integration of these concepts into a working system seems to refute this negative opinion. On one hand, the need for semantic simplification invariably arises when higher-level interfaces such as natural language [Ott and Horlaender 1982] are to be implemented that rely heavily on view mechanisms.

An important if trivial observation in this context is that -- in contrast to integrity checking in update operations -- the query simplifier has complete freedom to use just as many constraints as justified by the expected benefit. The modular implementation enabled by logic programming in connection with the blackboard concept is particularly flexible in allowing the easy addition of 'expert rules' for which constraints to use in a given environment. For example, the current implementation tries to avoid the exponential search incurred by full handling of notequal conditions [Rosenkrantz and Hunt 1980] by ignoring certain notequal-related simplification strategies. Similarly, the initial shortest-path procedure currently appears to be the major performance bottleneck. We are therefore experimenting with 'expert rules' that reduce the number of inequalities based on valuebound conditions, and thus the number of nodes in the algorithm based on 'reasonable' -- but not failproof -- assumptions.

On the other hand, the implementation of the simplifier has demonstrated another, quite surprising advantage (although it may seem obvious in hindsight): the capability of the system to provide meaningful warnings in cases where previous query evaluation subsystems would just return an empty result. The need for such enhanced feedback was especially felt during our earlier work on empirically evaluating a natural language query system where users were often helpless when the system returned an unexpectedly empty result [Jarke et al. 1985].

Apart from our work on an improved interface from Prolog to the simplifier (handling recursion and buffer management [Jarke et al. 1984]), two extensions to the simplifier itself seem particularly promising. The first is the analysis and optimization of predicates handling arbitrary functions over database data which should lead to improved database interfaces to decision support systems, statistical databases, recursive databases, etc.

Additionally, work is underway to extend the simplifier to the multi-user case. This idea is presumed to have several advantages. First, since all users would be read-only, the simplifier requires only rudimentary concurrency control and can thus be a relatively small and simple system. Second, since the simplifier is external, it can interact with the DBMS as a single user, thus reducing DBMS concurrency control problems. Third, as a consequence of the previous two, the simplifier has full freedom to perform common subexpression analysis to share query evaluation costs and to create common temporary access paths [Jarke 1984]. Finally, as a consequence of its global architecture (section 3), the simplifier can easily accept multiple input languages, although, from the viewpoint of error messages and efficient common access path analysis, a single input language, e.g., Prolog, may be more desirable since it would allow the addition of view definitions to the knowledge base.

In summary, it appears that narrowing the scope of semantic query optimization to database theory-based simplification -- while keeping the general idea in mind -- has some benefits of simplicity and efficiency. This should by no means be constructed as a criticism of general semantic query optimization. On the contrary, we see our approach as a kernel around which more sophisticated knowledge bases can be constructed, whose corresponding inference techniques work on the same blackboard data structure, hopefully with little interference with existing algorithms. Further classification of integrity constraints may be desirable for such extensions; in particular, those types of constraints should be investigated for which the range of applicability is easily detectable and does not, in itself, require answering a complex query.

Acknowledgments. The concept of the external semantic query simplifier builds on earlier work with Juergen Koch and Joachim Schmidt on query transformation strategies in database programming languages, and with Jim Clifford and Yannis Vassiliou on interfacing Prolog with relational database systems.

References

1. Aho, A.V., Sagiv, Y., Ullman, J.D., "Equivalences among relational expressions", SIAM Journal of Computing 8, 2 (1979), 218-246.

2. Blaustein, B.T., "Enforcing database assertions: techniques and applications", Ph.D. thesis, Harvard 1981.

3. Cosmadakis, S.S., Kanellakis, P.C., "Functional and inclusion dependencies: a graph-theoretic approach", Proceedings ACM-PODS Conference, Waterloo 1984, 29-37.

4. Downey, P.J., Sethi, R., Tarjan, R.E., "Variations on the common subexpression problem", Journal of the ACM 27,4 (1980), 758-771.

5. Du, G.D., "Search control in semantic query optimization", TR# 83-09, University of Massachusetts, Amherst 1983.

6. Erman, L.D., Lesser,V.R., "A multi-level organization for problem solving using many diverse, cooperating sources of knowledge", Proceedings 4th IJCAI Conference, 1975, 483-490.

7. Fagin, R., "A normal form for relational databases that is based on domains and keys", ACM Transactions on Database Systems 6, 3 (1981), 387-415.

8. Hammer, M., Zdonik, S.B., "Knowledge-based query processing", Proceedings 6th VLDB Conference, Montreal 1980, 137-147.

9. Jarke, M., "Common subexpression isolation in multiple query optimization", in W.Kim, D.Reiner, D.Batory (eds.): Query Processing in Database Systems, Springer-Verlag, to appear 1984.

10. Jarke, M., Clifford, J., Vassiliou, Y., "An optimizing Prolog front-end to a relational query system", Proceedings ACM-SIGMOD Conference, Boston 1984, 296-306.

11. Jarke, M., Koch, J., "Range nesting: a fast method to evaluate quantified queries", Proceedings ACM-SIGMOD Conference, San Jose 1983, 196-206.

12. Jarke, M., Sivasankaran, T., "Knowledge-based model management in an actuarial consulting systems", Proceedings 6th European Conference on Artificial Intelligence, Pisa, September 1984.

13. Jarke, M., Turner, J.A., Stohr, E.A., Vassiliou, Y., White, N., Michielsen, K., "A field evaluation of natural language for data retrieval", IEEE Transactions on Software Engineering, to appear 1985.

14. Jarke, M., Vassiliou, Y., "Coupling expert systems with database management systems", in Reitman, W. (ed.), Artificial Intelligence Applications for Business, Ablex, Norwood, NJ, 1984, 65-85.

15. Kellogg, C., "A practical amalgam of knowledge and data base technology", Proceedings National Conference on Artificial Intelligence, Pittsburgh 1982.

16. King, J.J., "QUIST: A system for semantic query optimization in relational data bases", Proceedings 7th VLDB Conference, Cannes 1981, 510-517.

17. Klug, A., "Calculating constraints on relational expressions", ACM Transactions on Database Systems 5, 3 (1980), 260-290.

18. Nicolas, J.-M., Yazdanian, K., "An outline of BDGEN: A deductive DBMS", in R.E.Mason (ed.), Information Processing 83, North-Holland 1983, 711-717.

19. Ott, N., Horlaender, K., "Removing redundant join operations in queries involving views", IBM Scientific Center Heidelberg Technical Report TR-82.02.003 (1982).

20. Reingold, E.M., Nievergelt, J., Deo, N., Combinatorial Algorithms. Theory and Praxis, Prentice Hall 1977.

21. Rosenkrantz, D.J., Hunt, M.B. "Processing conjunctive predicates and queries", Proceedings 6th VLDB Conference, Montreal 1980, 64-74.

22. Vassiliou, Y., Jarke, M., Clifford, J., "Expert systems for business applications: a research project at New York University", IEEE Database Engineering 6, 4 (1983), 50-55.

23. Vassiliou, Y., Clifford, J., Jarke, M., "Access to specific declarative knowledge by expert systems: the impact of logic programming", Decision Support Systems 1, 1 (1984).

24. Warren, D.H.D., "Efficient processing of interactive relational data base queries expressed in logic", Proceedings 7th VLDB Conference, Cannes 1981, 272-282.